Robust Linux Socket

VC IVI Development Center Vietnam

Hanoi, September 2018

Content

- Choosing when wait for I/O
  - I/O Blocking: Why?
  - When Do I Block?
  - Alternatives to I/O Blocking
  - Comparing the Different I/O Programming Interactions
  - Polling I/O
  - Asynchronous I/O
  - Solving Unwanted I/O Blocking with poll() and select()
- Dividing the load: multitasking
- *Breaking Performance Barriers*
  - *Creating the Servlets Before the Client Arrives*
  - *Extending Control with a Smart Select*

I/O Block: why ?

- Every task has to yield time to other tasks: each task gets a slice of time from the CPU. If there is nothing to do, the task yields up its slice.
- The primary cause of having nothing to do is waiting for some I/O to complete (speed of the disk or network is far lower than the speed of the processor)
  - .i.e a network delay of 20ms could mean that your task could have executed about 5,000,000 opcodes, or 500,000 lines of C code
  - To check running task on your computer, run “ps aux”

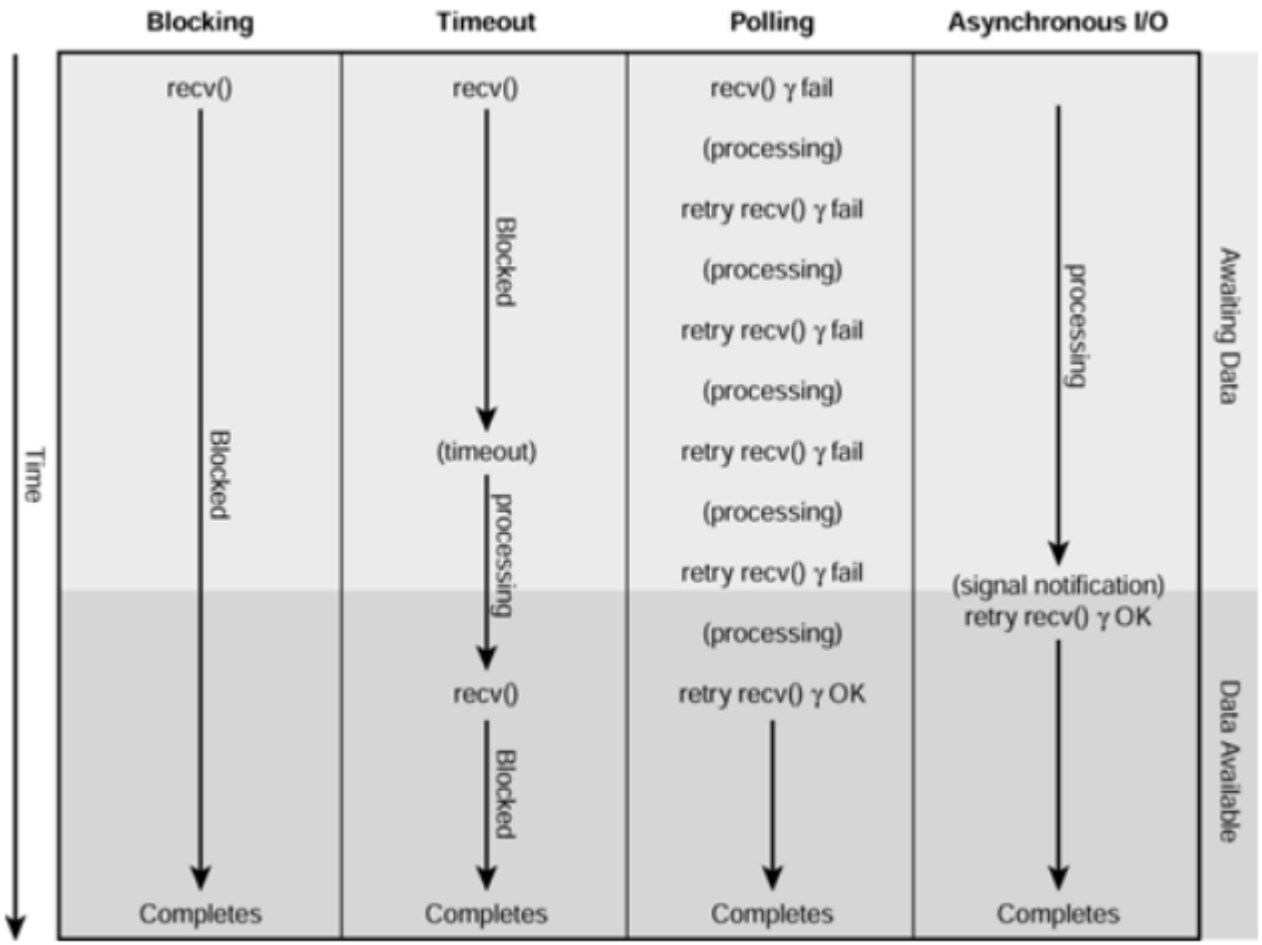
When do I block ?

- A program block (stop processing and wait) when
  - an I/O resource takes its time to finish
  - Any time your program makes a system call
- Sample circumstances
  - Reading — A read-block occurs when no data has arrived yet.
  - Writing — A write-block occurs when the internal buffers are full and waiting for transmission and your program requests more data to be sent.
  - Connecting — A connection-block occurs when the accept() and connect() system calls find no pending connections in the listening queue.

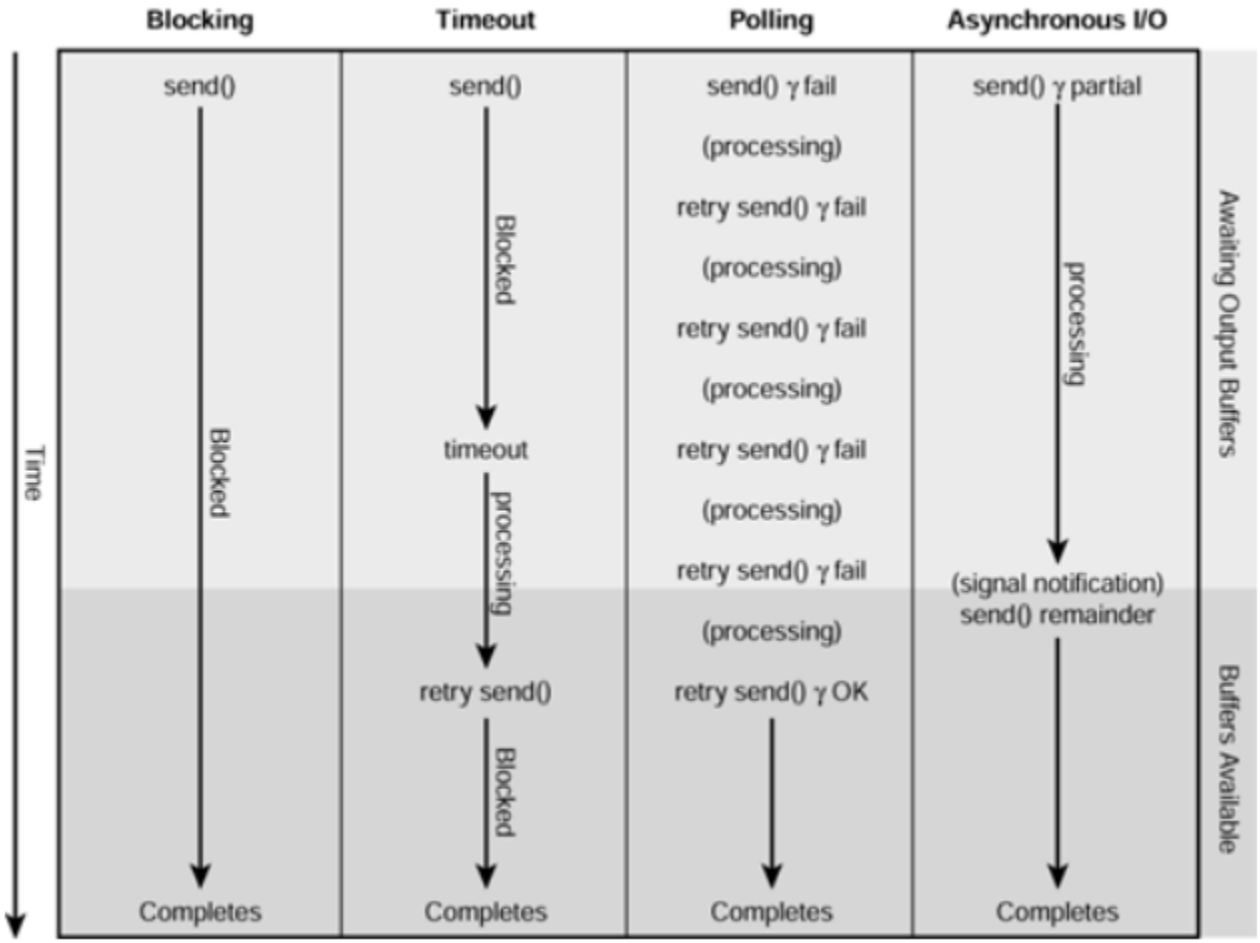
Alternatives to I/O Blocking

- What are some alternatives to blocking that you can use?
  - Test the integrity of its data.
  - Start and track other requests.
  - Wait for several socket connections.
  - Process some CPU-intensive calculations
- Three methods to help avoid blocking: **polling**, **timeouts**, and **asynchronous I/O**

Different I/O Programming Interactions [1]



Different I/O Programming Interactions [2]



Polling I/O [1] – configure nonblocking

- To use polling, configure the channel for nonblocking

```
#include <fcntl.h>
int fcntl(int fd, int command, int option);
```

  - *fd*: either a file descriptor (fd) or a socket descriptor (sd)
  - *command* is F_SETFL (for setting file status flags)
  - *option* is O_NONBLOCK

Polling I/O [2] - read example

```
/*****************************************************************/
/*** Polling Read Example: reading an audio stream,            ***/
/*** processing the data, and playing it.                      ***/
/*****************************************************************/
…
if ( fcntl(sd, F SETFL, O NONBLOCK) != 0 )
    perror("Fcntl--could not set nonblocking");
…
done = 0;
while ( !done )
{    int bytes;
    Queue ProcessingQueue;
    Queue OutputQueue;
   /*--- Get any pending data ---*/
    if ( (bytes = recv(sd, buffer, sizeof(buffer), 0)) > 0 )
        QueueData(ProcessingQueue, buffer);
   /*--- Convert a certain number of bytes from the ---*/
    /*--- processing queue--typically faster than recv() ---*/
    ConvertAudio(ProcessingQueue, OutputQueue);
   if ( /*** output queue is far enough ahead ***/ )
        PlayAudio(OutputQueue);
   /*--- If data stream is done & output queue empty… ---*/
    if ( bytes == 0  &&  /*---output queue empty---*/ )
        done = 1;
}
…
```

Polling I/O [3] - writer example

```
/*****************************************************************/
/*** Polling Writer Example: sending an image to several  ***/
/*** clients.                                             ***/
/*****************************************************************/
…
int pos[MAXCLIENTS];
bzero(pos, sizeof(pos));
for ( i = 0; i < ClientCount; i++ )

    if ( fcntl(client[i], F_SETFL, O_NONBLOCK) != 0 )
        perror("Fcntl--could not set nonblocking");
…
done = 0;
/*--- repeat until all the clients get the whole message ---*/
while ( !done )
{    int bytes;
   done = 1;
    /*--- For all clients… ---*/
    for ( i = 0; i < ClientCount; i++ )
        /*--- If still more to send to client… ---*/
        if ( pos[i] < size )
        {
            /*--- send msg, tracking how many bytes sent ---*/
            bytes = send(client[i], buffer+pos[i],
                size-pos[i], 0);
            if ( bytes > 0 )
            {
               pos[i] += bytes;
                /*--- if all the clients are done, exit ---*/
                if ( pos[i] < size )
                    done = 0;
            }
        }
}
…
```

Polling I/O [4] polling connection example

```
/*************************************************************/
/*** Polling Connection Example: check various ports for  ***/
/*** connections creating a new task to answer each       ***/
/*** request.                                             ***/
/*************************************************************/
…
/*--- Set nonblocking for each socket ---*/
for ( i = 0; i < numports; i++ )
    if ( fcntl(sd[i], F_SETFL, O_NONBLOCK) != 0 )
        perror("Fcntl--can't set nonblocking on port#%d", i);
…
for (;;)                                    /* Do this forever */
{    int client;
   for ( i = 0; i < numports; i++ )
        if ( (client = accept(sd[i], &addr, &size)) > 0 )
            SpawnServer(sd[i], i);
    /**** Do housekeeping ****/
}
…
```

Asynchronous I/O

- The system implements this asynchronous I/O with the SIGIO signal (sometimes called *signal-driven I/O*): Programs get a SIGIO when data is ready for reading or when a write channel is ready for more data

```
/*************************************************************/
/*** General Asynchronous or Signal-Driven I/O Algorithm  ***/
/*************************************************************/
int ready=0;
…
void sig_io(int sig)
{
    /**** IF RECV(): Get all pending data ****/
    /**** IF SEND(): Send all processed data ****/
    ready = 1;     /* tell main loop: "transaction complete" */
}
…
for (;;)
{
    if ( ready > 0 )
    {
        /**** Block SIGIO temporarily ****/
        ready = 0;
        /**** IF RECV(): copy data into processing area ****/
        /**** IF SEND(): fill output buffer from ****/
        /****    …processed data queue ****/
        /**** Unblock SIGIO ****/
    }
    /**** Process incoming data ****/
    /**** -OR- ****/
    /**** Generate more data to send ****/
}
…
```

Turning off the signal handler is an important mutual exclusion device (mutex) because the signal handler and the main loop share the same variables

Solving Unwanted I/O Blocking with poll() and select()

- Linux offers two tools (poll() & select()) to help you work with multiple open channels at the same time
  - system call blocks until any channel changes state
  - A state change includes conditions such as data being available to read, a channel is clear for writing, or an error occurred
  - The return value represents the number of channels that changed

Using select() [1]

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int maxfd, fd_set *to_read, fd_set *to_write,
    fd_set *except, struct timeval *timeout);
FD_CLR(int fd, fd_set *set);          /* remove fd from set */
FD_ISSET(int fd, fd_set *set);     /* check if fd is in set */
FD_SET(int fd, fd_set *set);              /* add fd to set */
FD_ZERO(fd_set *set);             /* initialize set for use */
```

Parameters for the `select()` System Call

Parameter	*Description*
`maxfd`	One more than the highest-numbered file descriptor in the set. (See below for more.)
`to_read`	The set of descriptors that await reading. This is the set that you want to test for reading.
`to_write`	The set of descriptors that await writing.
`except`	The set of descriptors that await priority messages.
`timeout`	The time in microseconds to wait before giving up. (See below for more details.)
`fd`	The file descriptor to add, delete, or test for.
`set`	The set of descriptors to manipulate.

timeout parameter gives you the flexibility of setting a limit to how long the system call waits:

- If NULL, wait indefinitely.
- If positive, wait the specified microseconds.
- If zero, return immediately after checking all descriptors once

Using select() [2]

```
/*************************************************************/
/*** Select() Example.                                     ***/
/*************************************************************/
int count;
fd_set set
struct timeval timeout;
…
FD_ZERO(&set);                                  /* Clear set */
FD_SET(3, &set);                    /* Add client channel #3 */
FD_SET(4, &set);                    /* Add client channel #4 */
FD_SET(6, &set);                    /* Add client channel #6 */
timeout.tv_sec = 5;             /* Timeout after 5.25 seconds */
timeout.tv_usec = 250000;
/*--- Wait for select() ---*/
if ( (count = select(6+1, &set, 0, 0, &timeout)) > 0 )
    /**** Find the descriptors that changed ****/
else if ( count == 0 )
    fprintf(stderr, "Timed out!");
else
    perror("Select");
…
```

Using poll() [1]

```
#include <sys/poll.h>
int poll(struct pollfd *list, unsigned int cnt, int timeout);

struct pollfd
{
    int fd;         /* the descriptor to test */
    short events;   /* requested events to watch */
    short revents; /* events that fired */
} ;
```

The first field, `fd`, is the file or socket descriptor. The second and third fields, `events` and `revents`, are bit masks for specific events:

- `POLLERR` Any error condition. Return if any error occurs on channel.
- `POLLHUP` Hang-up on other end. Return if peer hangs up.
- `POLLIN` Incoming data. Return if data ready for reading.
- `POLLINVAL` `fd` not open. Return if the channel is not an open file or socket.
- `POLLPRI` Priority messages. Return if a priority message arrives.
- `POLLOUT` Channel availability. Return if a `write()` won't block.

Implementing Timeouts [1]

- ways to implement timeouts:
  - With the timeout in the select() or poll() system call
  - With a signal (SIGALRM) to wake up the task

  An *alarm* is a clock that runs in the kernel. When the alarm goes off, the kernel sends a wakeup call to the task in the form of a signal. If the task is waiting in a system call, the signal interrupts the call, the task enters the signal handler, and then the system call returns with an EINTR error

Implementing Timeouts [2]

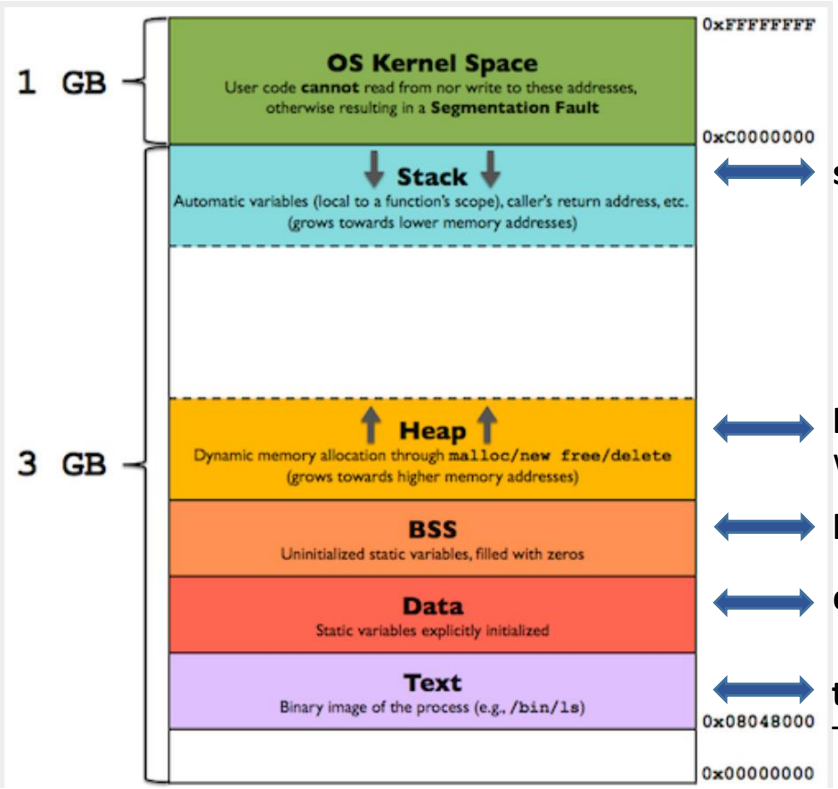
```
/*************************************************************/
/*** Example: Timeout example using the alarm() system     ***/
/*** call. (Excerpt from echo-timeout.c on CD.)            ***/
/*************************************************************/
…
int sig_alarm(int sig)
{ /*** do nothing ***/}
…
void reader()
{    struct sigaction act;
   /*--- Initialize signal structure; reset to default ---*/
    bzero(&act, sizeof(act));
    act.sa_handler = sig_alarm;
    act.sa_flags = SA_ONESHOT;
   /*--- Set the signal handler ---*/
    if ( sigaction(SIGALRM, &act, 0) != 0 )
        perror("Could not set up timeout");
    else
        /*--- If signal handler installed, set timer ---*/
        alarm(TIMEOUT_SEC);
   /*--- Call the I/O that might timeout ---*/
    if ( recv(sd, buffer, sizeof(buffer), 0) < 0 )

    {
        if ( errno == EINTR )
            perror("Timed-out!");
    }
}
…
```

Dividing the load: multitasking

Process and memory mapping

- Process

A process is a program in execution. When a operating system executes a program it is called a process. So we may have a single program executed simultaneously multiple times which creates multiple processes. Also we may have a single program which creates multiple processes in the program. Each process has a unique ID called the process ID allocated by OS.



stack: This is where the local variables and function calls are stored.

heap: This is where user allocated memory is stored.
When the user uses malloc, calloc, new etc.

bss: This is where global and static un-initialized variables are stored.

data: This is where global and static initialized variables are stored.
- **rodata:** This is where constant variables are stored.

text: This is where the code of the process reside.
This is a read-only section.

How a process looks like in main memory on a typical 32-bit Linux OS.

Fork & Thread

- FORK

Forking is nothing but creating a new process. We create a new process that copies all the elements of old process.

- THREAD

Threading is a light weight process which shares all the section of the process except for the stack. A process can have multiple threads.

How Do I Create a Process? [1]

- use the *fork*() system call

```
#include <unistd.h>
pid_t fork(void);
```

  - Zero—The program in question is the child. If the call returns a zero, the *fork*() was successful and the currently running task is the child. To obtain the child's process ID (PID), use the *getpid()* system call.
  - Positive—The program in question is the parent. If the call returns a positive value, the *fork*(), again, was successful and the currently running task is the parent. The return value itself is the PID of the new child.
  - Negative—An error occurred; the call was unsuccessful. Check errno or use *perror()* to determine the nature of the error

How Do I Create a Process? [2]

Task splitting example

```
/*****************************************/
/* Parent and Child following unique paths */
/* Task-splitting style of multitasking    */
/*****************************************/
int pchild;
if ( (pchild = fork()) == 0 )
{    /* You're in the child */
    /*---Do some child-ish things---*/
    exit(status); /* This is important! */
}
else if ( pchild > 0 )
{    /* You're in the parent */
    int retval;

    /*---Do some parent-ish things---*/
    wait(&retval); /* await child's end */
}
else
{    /* Some kind of error */
    perror("Tried to fork() a process");
}
```

The parent does some work and then waits for the child to terminate

Job delegation example

```
/*****************************************/
/* Parent (server) and Child (job proc)   */
/* Job-delegation style of multitasking   */
/*****************************************/
int pchild;
for (;;) /* Loop Forever */
{
    /*---Wait for job request---*/
    if ( (pchild = fork()) == 0 )
    {    /* Child */
        /*---Process the request---*/
        exit(status);
    }
    else if ( pchild > 0 )
    {
        /* parent clean up */
        /* notice NO WAIT() -- you must */
        /* use signals (see later) */
    }
    else if ( pchild < 0 )
    {    /* Some kind of error */
        perror("Can't process job request");
    }
}
```

The parent works as a job delegator: when some outside entity requests an operation, the parent creates a child to do it. Most servers use this technique.

How Do I Create a Thread? [1]

- use library call *pthread_create*()

```
#include <pthread.h>
int pthread_create(pthread_t* child, pthread_attr_t* attr, void*
(*fn)(void*), void* arg);
```

  - The *pthread_create()* is in a different library that translates the call into the *__clone*() system call
  - To compile programs with threads, be sure to include -lpthread as the last argument on the cc command line
    - cc mythreads.c –o mythreads **–lpthreads**

How Do I Create a Thread? [2]

```
#include <pthread.h>
int pthread_create(pthread_t* child, pthread_attr_t* attr, void*
(*fn)(void*), void* arg);
```

The Function of Each Parameter for the `pthread_create()` Library Call

Parameter	*Description*
`child`	The handle to the new thread. You use this variable after the call to control the thread.
`attr`	A set of attributes or behaviors for the new thread. You can define how the thread is to work and interact with the parent at the time of instantiation. A few of these behaviors include priority, attachment, and scheduling. This parameter can be `NULL`.
`fn`	A pointer to a function that is to house the thread. Every thread runs in a specified program subroutine (unlike `fork()`). When the thread returns or exits the subroutine, the system terminates the thread. This helps avoid the task confluence problem (see above).
`arg`	A parameter to be passed to the function. You can use this to configure the thread's initial settings. Be sure to make the data block that this parameter points to available to the thread. In other words, don't refer to a stack variable. The parameter can be `NULL` as well.

As previously noted, you have two threads running after successfully completing the call—the parent and the child. Both share everything except the stack.

How Do I Create a Thread? [3]

Process creation example

```
/* This example creates a process */
/*********************************/
void Child_Fn(void)
{
    /* do whatever a child does */
}
int main(void)
{    int pchild;
   /*---Initialization---*/
    /* Create new process */
    if ( (pchild = fork()) < 0 )
        perror("Fork error");
    else if ( pchild == 0 )
    { /* here is the child */
        /* close unneeded I/O */
        Child_Fn();
        exit(0);
    }
    else if ( pchild > 0 )
    { /* Here is the parent */
        /*---close unneeded I/O---*/
        /*---do other work---*/
        /* wait for child to end */
        wait();
    }
    return 0;
}
```

Processes require a specific call to ***exit()*** to avoid running together, which you place at the end of the child's path

Thread creation example

```
/*********************************/
/* This example creates a thread */
/*********************************/

void *Child_Fn(void *arg)
{    struct argstruct *myarg = arg;
    /* do whatever a child does */
    return NULL; /* any value */
}
int main(void)
{    struct argstruct arg = { } ;
    thread_t tchild;
   /*---do some other initialization---*/
    /* Create new thread; report error in failed */
    if ( pthread_create(&tchild, NULL, &Child_Fn, &arg) != 0 )
        perror("PThreads error"); /* error */
   /***Note that threads don't have a child section***/
    /* We are implicitly still in the parent */
   /* do some other work */
    /* wait for child to end */
    pthread_join(tchild, NULL);
   return 0;
}
```

With Pthreads, you don't have to worry about that. A return statement (or even falling through without a return statement) implicitly terminates the child thread

Demo

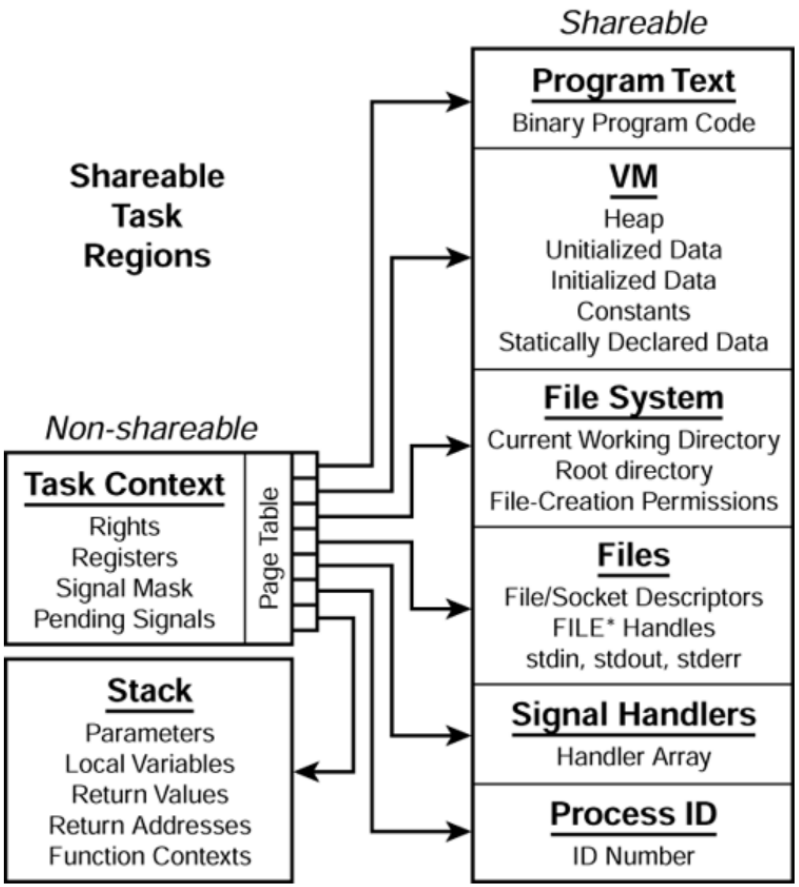
- forking.c
- threading.c
- echo-process.c — socket server (using fork())
- echo-thread.c — socket server (using thread)
- simple-client.c — socket client to connect to server

History

Change History			
Revision	Date	Author	Comments
0.1	01.11.2021	Nam2.nguyen	Initial version


THANKSess.
FOR YOUR ATTENTION
memegenerator.net

Appendix



Appendix: Process vs Threads

Processes	*Threads*
From the point of a successful call, there are two processes running inline.	The caller supplies the name of the function to run as the child.
The child must be explicitlyterminated with an `exit()` system call.	The child can explicitly or implicitly terminate with a `pthread_exit(void* arg)` or a return statement.
	No shared data. The only information passed to the child is the snapshot of the parent's data before the system call. The child shares the parent's data, accepts a parameter, and/or returns a value.
The child is always associated with the parent. When it ends, the parent must clean up the child.	The child can be detached from the parent. Thus, the child can terminate without parental intervention.(Unless the child is detached, the parent has to clean up the thread as well.)
Because each process's data is independent from the others, there is no contention for resources.	All contended data must be identified and locked so no corruption can occur.
Independent file system.	Linked file systems. The child sees any changes the parent makes to the current working directory (`chdir`), the root file system (`chroot`), or the default permissions on file creation (`umask`).
Shared file tables. If the child closes a file, the parent loses access to the file.	Open file tables are not shared. The operating system copies the tables so that the two processes have the same file open, but closing one channel does not affect the other process.
No shared signals.	Signals are shared. One thread can block a signal using `sigprocmask()` without affecting the other threads.

This simple rule to decide between processes or threads:
If you need to share data, use a thread